Execute-Only Attacks against Execute-Only Defenses[1]

Robert Rudd*, Thomas Hobson*, David Bigelow*, Richard Skowyra*, Veer Dedhia*,
Stephen Crane†, Per Larsen†, Andrea Homescu†, Christopher Liebchen‡,
Lucas Davi‡, Ahmad-Reza Sadeghi‡, Michael Franz†, William Streilein*, Hamed Okhravi*
*MIT Lincoln Laboratory
†University of California, Irvine
‡TU Darmstadt

***Abstract*—Execute-only defenses have been proposed as a way of mitigating information leakage attacks that have been widely used to bypass randomization-based memory corruption defenses. A recent technique, Readactor, provides one of the strongest implementations of execute-only defenses: it exploits novel hardware features to incorporate non-readable code to prevent direct information leakage, a layer of indirection to prevent indirect information leakage of pointers located on stack and heap, and code randomization as well as decoys to prevent brute-force attacks. In this paper, we demonstrate three novel attacks that can bypass Readactor as well as numerous other recent memory corruption defenses with various impacts. We analyze the prevalence of opportunities for such attacks in popular code bases and build two proof-of-concept exploits. Moreover, we implement countermeasures against our attacks in Readactor itself and discuss their implications. Our evaluations indicate that our countermeasures introduce only a modest additional overhead.**

I. Introduction

Memory corruption has been a primary vector of attacks against diverse computer systems for decades [4]. From conventional stack smashing techniques [35] to the more sophisticated code-reuse attacks (e.g., return-oriented programming, abbr. ROP) [44] devised as a result of widespread adoption of defenses such as W⊕X (Write⊕eXecute) [36]. Part of the appeal of memory corruption for attackers is its ability to remotely control the target system in what is known as control hijacking attacks [1]. Despite numerous advances in this domain, low-overhead techniques that protect unmanaged languages such as C/C++ against memory corruption is still an open area of research [48].

Two classes of memory corruption defenses have been widely studied in the literature: enforcement-based defenses and randomization-based. In the enforcement-based class, a policy is enforced on the program's execution flow at runtime. Control-flow integrity [1] and its variants [30, 53, 52], and memory safety checks [32] are all examples of enforcement-based defenses. In the randomization-based class, the code or its execution is randomized or diversified [28] to make it unknown for an attacker. Address Space Layout Randomization (ASLR) [38], binary rewriting [49], compiler-based diversity [22], and memory re-randomization [7] are all examples of randomization-based defenses.

A class of attacks that is particularly damaging to randomization-based defenses is information leakage attacks [46]. Using information leakage, an attacker can discover how code or its layout has been randomized, which in turn, undoes the impact of randomization. Direct leakage of memory content (a.k.a., memory disclosure) [46], indirect leakage of addresses from the stack or heap [15], and remote side-channel attacks [42] are different forms of information leakage that have been used successfully to bypass recent randomization-based defenses [16, 15, 11]. Due to the prevalence and power of such information leakage attacks, recent defenses assume arbitrary read and write capability to memory locations for attackers in their threat model if page permissions allow such operations [13].

In response to information leakage attacks, researchers have developed "execute-only" defenses that prevent "read" operations on code regions [5, 13, 18]. The enforcement of R⊕X (Read⊕eXecute) also known as XnR (eXecute, no Read) prevents direct information leakage from memory [5]. Readactor [13], is one of the most effective, state-of-the-art XnR defenses that combine execute-only permissions on code pages (using Intel's Extended Page Table permissions) to mitigate direct information leakage, redirection of indirect branches through a set of *trampolines* to mitigate indirect leakage of addresses during execution (a.k.a. *code pointer hiding*), and randomization of trampolines to create uncertainty for an attacker. A recent extension to Readactor, called Readactor++ [14], also incorporates decoy trampolines and register randomization to prevent control hijacking through brute-force by an attacker that can leak large parts of memory. It also protects C++ data structures such as vtables to mitigate COOP attacks that inject counterfeit objects into memory [40]. Unless otherwise noted, in the rest of this paper we refer to both Readactor and Readactor++, simply as Readactor. Other XnR defenses have also been proposed in the community, but they can only mitigate a subset of attacks mitigated by Readactor. For example, the original XnR work [5] and HideM [18] can only resist direct information leakage, while they are still vulnerable to indirect leakages or remote side-channel

[1]Distribution A: Public Release. This work is sponsored by the Department of Defense under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions and rec-ommendations are those of the author and are not necessarily endorsed by the United States Government.

attacks.

Contributions. In this paper, we describe and demonstrate two classes of attacks that can bypass execute-only defenses including Readactor. The first class leverages the information leaked to an attacker during execution to bypass execute-only defenses, even when execute-only is implemented perfectly and comprehensively. We show that an attacker can leak trampoline addresses from the stack during execution in what we call *Address Harvesting (AH)*. In addition, we devise new techniques for performing address harvesting accurately, chaining trampolines together, and implementing code reuse attacks at the trampoline granularity. We generalize the COOP attack [40] to non-Object Oriented Programming languages to create a chaining mechanism for the trampolines.

The second class of attacks exploit the fact that enforcing ideal and comprehensive execute-only defense is actually challenging in modern systems. We generally call these attacks *Execute-Only Bypasses* and demonstrate two practical attack vectors in Linux systems. The first vector maliciously redirects Direct Memory Access (DMA) operations that do not abide by page permissions. The second vector uses Linux's Proc filesystem to directly leak memory content. Both these vectors can be used to maliciously leak actual non-readable code pages after which traditional ROP attacks become straightforward.

We build two proof-of-concept exploits against Nginx combining these various techniques to achieve control hijacking in the presence of Readactor. Moreover, these attacks are not limited to Readactor; we discuss the generality of these attacks against other recent defenses and show that many of them are vulnerable to the same exploits.

Using the lessons learned from these attacks, we build and evaluate countermeasures against them. To prevent the Address Harvesting attack, we build a technique call Indirect Branch Authentication that prevents trampoline-reuse attacks. We augment Readactor, itself, to prove the feasibility and effectiveness of our technique. To prevent the execute-only bypasses, we augment the lightweight hypervisor used by Readactor with an IOMMU-like functionality to extend the page permissions to DMA operations as well. Preventing the Proc filesystem attack is difficult in the general case because its removal will break many benign applications. We present an analysis of popular code bases that require access to Proc filesystem and discuss the implications of various approaches to mitigate the attack.

In summary, our contributions are as follows:

- We present two classes of attacks; one that can bypass an ideal execute-only defense and one that exploits the practical limitations of such defenses. We build two proof-of-concept exploits that can achieve control flow hijacking on a system protected by full-featured Readactor.
- We evaluate the prevalence of opportunities for our attacks in popular code bases and show that there is an abundance of vulnerable cases in real world.
- We discuss the generality of our attacks against other recent defenses and demonstrate that they can successfully bypass many of them, highlighting the intricacies of effective memory corruption defenses.
- We propose and implement countermeasures against our attacks including Indirect Branch Authentication. We augment Readactor with our countermeasure and show that it can mitigate trampoline reuse and similar attacks.
- We evaluate the impact of our new countermeasures and demonstrate that they introduce modest additional overhead.

Section II describes our threat model. Section III provides an overview of the two classes of attacks. Sections IV and V present the details of each class of attack. In Section VII, we discuss the applicability and implications of our attacks against recent defenses. Section VIII describes the countermeasures against our attacks and their implementation. Section IX provides the evaluation results assessing the impact of the countermeasures. We review the related work in Section X before concluding the paper in Section XI.

II. Threat Model

Our threat model assumes a remote attacker that exploits a memory vulnerability to achieve remote code execution on the target machine. We assume W$\oplus$X is deployed, so code cannot be modified and data cannot be executed. Moreover, we assume Readactor and Readactor++ are deployed to protect the target application. Although, Readactor and similar techniques in the literature assume arbitrary read and write accesses into memory for the attacker (if page permissions allow them), we show that practical attacks are possible even when weaker capabilities are assumed. In practice, for the Address Harvesting attack, we only need multiple read accesses into stack at specific times. Other attacks presented in this paper (such as the Forged DMA attack) have a more detailed threat model which will be presented in their respective sections.

On the other hand, for our indirect branch authentication extension to Readactor, we assume the strongest threat model which includes the presence of memory vulnerabilities and arbitrary attacker read and write access to memory.

III. Background and Overview of Attacks

In this section we provide an overview of our attacks against Readactor. We discuss these attacks in detail in the subsequent sections. Readactor aims to prevent code reuse attacks by limiting an attacker's ability to learn the locations of existing code, either by leaking the code itself or by leaking pointers to the code. Readactor operates under a strong threat model in which an adversary possesses arbitrary read and write vulnerabilities. In defending against such an adversary, two primary techniques are employed by Readactor:

- *Execute-only code* enforces execute-only permission on code pages. Thus, any attempts by an attacker to learn the locations of code by leaking the contents of code pages directly will fail due to the lack of read permission.
- *Code pointer hiding via indirection* modifies the code such that no pointers to the original code (i.e. function pointers or return addresses) will reside in data segments.

It achieves this by replacing calls to the original code with calls to trampoline code; the trampoline code references the original code directly. The result is that the only pointers to code which can reside in memory are pointers to the trampoline code.

We show attacks, called Address Harvesting, that leak trampoline addresses even when ideal execute-only is enforced to perform a form of indirect code reuse we call Trampoline-Oriented Programming and attacks that use the intricacies of comprehensive execute-only enforcement to directly leak code in execute-only regions to perform traditional code reuse attacks.

A. Address Harvesting

A Trampoline-Oriented Programming (TOP) attack can be performed even if execute-only permissions are applied ubiquitously throughout the system. A TOP attack involves various stages. In the first stage, we use Address Harvesting to reveal trampoline pointers rather than pointers to the original code itself. We profile the trampoline pointers to map them to the underlying functions invoked by the trampolines and launches a code reuse attack using these trampolines rather than the underlying functions themselves. The layer of indirection provided by the trampoline only serves to push the problem off rather than eliminate it. We demonstrate a technique called Malicious Thread Blocking (MTB) that allows an attacker to harvest addresses with high precision. This technique modifies mutexes to force a threat to block on certain inputs which allows us to harvest correct addresses while avoiding unnecessary application crashes. We discuss the details of these attacks in Section IV and show a proof-of-concept exploit using such techniques against Nginx in Section VI.

B. Execute-Only Bypasses

An execute-only bypass attack abuses the fact that ideal enforcement of execute-only permissions can be quite challenging in modern system. Unlike the TOP attack, execute-only bypasses attack the imperfections of execute-only defenses.

The most straightforward way to bypass Readactor is to leak the contents of code directly. Readactor implements execute-only memory by installing a hypervisor that checks the page permissions in Extended Page Tables (EPT) [21], a form of nested paging, during address translation. Unfortunately, the page permissions in EPTs are not enforced universally across memory read operations. We have identified two such instances in which page permission verification can be bypassed and execute-only memory can be read: the Direct Memory Access and Proc Filesystem.

1) Forged DMA: The first vector that we identified in which page permissions are not checked is in the case of Direct Memory Access (DMA). DMA provides a means to access memory quickly and independently of the processor. When DMA is used, memory contents are not copied into a kernel buffer and execute-only permissions are not checked. Files that are opened with the `O_DIRECT` flag will access memory

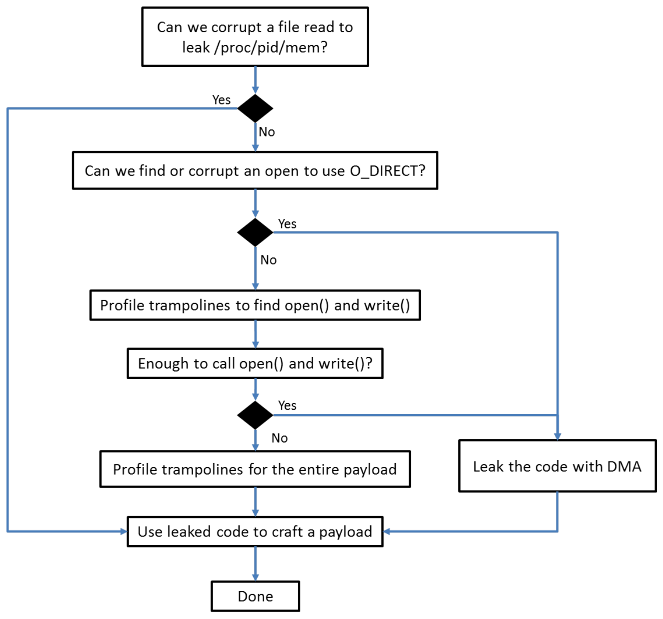

Can we corrupt a file read to leak /proc/pid/mem?
Yes
No
Can we find or corrupt an open to use O_DIRECT?
Yes
No
Profile trampolines to find open() and write()
Enough to call open() and write()?
Yes
No
Profile trampolines for the entire payload
Leak the code with DMA
Use leaked code to craft a payload
Done

Fig. 1. Possible paths to achieve code execution against Readactor

via DMA during I/O operations (i.e. `read()` and `write()`) and these operations will not be subject to the permission checks. Thus an attacker could modify the buffer passed to a `write()` call, in what we call a Forged DMA (FDMA) attack, in order to write arbitrary memory to an accessible target file descriptor (e.g. index.html for a web server). It is important to note that a program need not natively use the `O_DIRECT` flag as the attacker can overwrite the flags argument passed to the open() call using a simple data-only attack. In Section V-A we demonstrate this attack against SFTP.

2) Proc Filesystem: The `/proc/pid/mem` file contains the entire contents of a process' memory and its access does not trigger a check for execute-only permission. If a vulnerable program performs a file read operation an attacker could overwrite the path argument to point to `/proc/self/mem` in order to coerce the program to read the contents of its memory instead. If the attacker cannot read `/proc/self/mem`, she also has the option of leaking pointers to code locations from several other files within the Proc filesystem, such as `/proc/self/maps`, as discussed in Section V-B.

Figure 1 summarizes the paths that an attacker can take in order to achieve arbitrary code execution on a Readactor-protected binary. The simplest means is to overwrite the path argument to a file read to point to /proc/self/mem in order to read the contents of memory directly. Alternatively, if an attacker can control the arguments to the `open()` and `write()` calls, she could write the contents of memory to a file. If neither of these methods are available the attacker must proceed to Address Harvesting in order to determine the underlying functions for the trampoline. The trampolines can be harvested simply to the point of enabling one of the previously mentioned attacks or can be harvested to the extent of launching a complete code reuse attack using trampolines alone (i.e., TOP attack).

IV. Address Harvesting Attack

In this section, we describe a code reuse attack that circumvents Readactor and other execute-only memory defenses even under the strong assumption that execute-only permissions are universally enforced.

In addition to the execute-only permissions, Readactor creates a layer of indirection in order to prevent code pointers from residing in readable memory. At compilation time, Readactor ensures that any pointers to the original code of a program are moved into a trampoline section of execute-only memory such that only the trampolines directly reference the original code. Readactor effectively replaces the original pointers that were in readable memory (i.e. function pointers and return addresses) with pointers to trampoline code. Under this approach, a memory leak can only reveal the addresses of trampoline code, as opposed to the original program code itself. The underlying assumption of this indirection is that the attacker cannot utilize the trampoline pointers themselves to execute a useful code reuse attack.

We show how an attacker can indeed use these trampoline pointers to launch meaningful exploits. This is achieved by harvesting the trampoline pointers to determine the underlying original code to which they point. We demonstrate how multiple harvested trampoline pointers can be used together to launch a chained attack akin to traditional ROP; a Trampoline-Oriented Programming (TOP) attack.

A. Address Harvesting

The goal of harvesting is to determine the original function `f` that is invoked by a trampoline pointer `tptr`. An attacker who can identify the mapping of `tptr -> f` can redirect control flow to `f` in an indirect way via `tptr`. "->" denotes that `tprt` is the pointer to the trampoline that corresponds to function `f`.

To infer this mapping we exploit the fact that programs execute in a manner that inherently leaks information about the state of execution [42]. Knowledge about the execution state of a program at the time of a memory disclosure enables us to infer the `tptr -> f` mapping from a leaked `tptr`.

At the time a given disclosure vulnerability is triggered, memory contains pointers corresponding to the code on the vulnerable execution path leading to the vulnerability. An attacker, possessing knowledge about the execution path leading to the vulnerability, is similarly aware of which stack frames should exist on the stack and which function pointers (or trampoline pointers in the case of Readactor) should exist in these stack frames, for instance. Thus, trampoline pointers resident in these stack frames or other observable memory at the point of triggering the vulnerability are mappable to underlying functions by an attacker. At the moment the disclosure is triggered, the attacker may know that memory contains trampoline pointers to functions of interest and in this simple case the attacker's harvesting stage is complete.

1) Malicious Thread Blocking: A naïve approach to create an accurate mapping of trampoline pointers to underlying functions can rely on repeated stack disclosures and precise timing of the leakage. However, since the state of the system changes very rapidly, this can result in inaccuracies in the mappings which eventually causes a crash at the exploitation time. To enhance the precision of the mapping, we devised a technique which we call Malicious Thread Blocking (MTB).

In the case of programs that utilize threading, as in the attack model defined by Readactor, we can employ MTB that enables an attacker to harvest a broader range of trampolines and avoids dependence upon strict timing requirements for triggering the disclosure vulnerability.

The approach of MTB is to use one thread, T_A, to cause another thread, T_B, to hang at an opportunistic moment by manipulating variables that cause T_B's execution to block, e.g. by maliciously setting a mutex. By opportunistically blocking a thread, we can more easily locate and map desired trampoline pointers. A memory disclosure vulnerability may be triggered in T_A that enables memory inspection at a known point in execution in T_B. Note that this technique gets around any timing unpredictability that the attacker may face when trying to trigger a disclosure in thread T_A at the appropriate time in execution for thread T_B.

As one example of this technique in practice, we show in Section VI how an attacker can set a mutex in Nginx to cause a thread to block upon returning from a system call. Triggering a memory disclosure vulnerability in another thread at any point after the system call enables the attacker to inspect a memory state that she knows contains trampoline pointers relevant for the system call.

To more easily distinguish one system call from another, the attacker can supply unique input and scan disclosed memory for that input. For instance, if the attacker wishes the profile the `open()` call, she may supply a unique file name as normal input to the program. Upon inspecting the stack of a blocked thread, the attacker would expect to find this unique value as an argument to the `open()` call. An attacker can continually block and unblock a thread by manipulating the mutex until this value is discovered in disclosed memory which indicates that the attacker has located the relevant frame for `open()`. As the attacker is relying upon information leaked via execution, disclosures of this type require that the program natively use the functions that the attacker wishes to reuse.

B. Hijacking Control Flow

After the attacker has mapped relevant trampoline pointers to the underlying functions, it is straightforward to direct control flow to the function. The program has already been recompiled in such a way that rather than using code pointers it will use trampoline pointers, making it easy to overwrite values with trampoline pointers that will later be invoked.

Consider the use of function pointers. A standard use of a function pointer might be compiled into x86_64 assembly as `pop %r11, call %r11`, where the value of

%r11 represents the address of the function itself, we call it printf() for this discussion. An attacker with a memory corruption vulnerability that wishes to call execve() instead of printf() would overwrite the memory location from which %r11 is popped; changing a printf() call to an execve() call, for instance.

In Readactor, that code will be recompiled into pop %r11, jmp tramp_ind_cs200. A trampoline, tramp_ind_cs200, is also generated by Readactor for this specific call site. This call site invokes the code specified in register %r11 and, upon return from printf(), jumps back to the call site 200. Rather than pointing to printf() directly, the value in %r11 points to another trampoline, tramp_printf, which jumps directly to printf(). Rather than storing the address of printf() in memory, Readactor instead stores the value of tramp_printf. Thus, to invoke execve() an attacker would replace the value of tramp_printf in memory with the value of tramp_execve. The code will pop that value into %r11, jump to the trampoline at tramp_ind_cs200, which calls %r11 (tramp_execve), jumping to tramp_execve, which finally jumps to execve(). We have effectively replaced overwriting &printf() with &execve() for overwriting &tramp_printf with &tramp_execve.

C. Chaining

An attacker wishing to chain multiple functions together faces another challenge under Readactor. After the call to printf() completes, the code is going to ret, which returns to the value pushed onto the stack at the last call instruction, tramp_ind_cs200. Tramp_ind_cs200 jumps back to the call site associated with that trampoline. A challenge arises in chaining multiple calls together since a function will return to the call site from which it was invoked. If the attacker can overwrite the return value before the function issues the ret, she can point it to another leaked call site. However, this approach requires continuous rewriting of memory in order to ensure that the return address is overwritten at some point during the function call (e.g. after the call is made and before the ret is executed).

To overcome this challenge we introduce a technique we call Counterfeit Procedural Programming (CPP), which operates at a higher-level of abstraction, similar to the COOP [40] technique, but which works for non object-oriented languages such as C. Specifically, we leverage a loop construct that invokes a sequence of function pointers (or trampoline pointers in the case of Readactor). In this way, control is always returned to the loop following the execution of any function. An attacker can overwrite the sequence of trampoline pointers with a set of malicious trampoline pointers, effectively offloading the chaining to the existing loop mechanism.

In order to prevent passing computation results in a chained attack, Readactor employs register randomization. However, the register randomization is only applied within functions themselves and not across function boundaries, lest it break ABI. Since this attack is effectively a function-level ROP attack it can circumvent any register randomization that happens within a function itself. All function arguments and return values will reside in the standard set of registers.

V. Execute-Only Bypasses

This section describes a class of attacks which we call *execute-only bypasses*. Unlike the harvesting attack which is applicable even when execute-only is applied universally, execute-only attacks target the imperfections of enforcing execute-only permissions in modern systems. We discuss two attack vectors that are widely available in x86 Linux systems. Similar vectors may exist in other operating systems and architectures. Rather than indicating weaknesses in a particular execute-only technique, these attack vectors are meant to highlight the intricacies and challenges of applying execute-only permissions universally. Without loss of generality, we discuss these attacks in the context of Readactor, but discuss their general applicability in Section VII.

A. Forged Direct Memory Access Attack

Readactor relies on the presence of nested paging, a mechanism for providing hardware-based support for multiple-levels of address translation. Nested paging is called *Extended Page Tables (EPT)* in Intel processors, while it is called *Rapid Virtualization Indexing (RVI)* in AMD ones. For consistency with previous Readactor publications, this paper will refer only to Extended Page Tables, however the concepts are equally applicable to Rapid Virtualization Indexing.

Readactor implements execute-only memory via a small hypervisor implementation written as a Linux kernel module. When the Readactor kernel module is loaded, it initializes a set of Extended Page Tables, sets up a Virtual Machine Control Structure that uses these EPT's, and executes a VM entry instruction. This kernel module marks a page of memory as execute-only by setting the execute bit for the page's mapping and clearing the read and write bits. Any code that is executed within a VM context must adhere to the permissions specified in the EPT. This policy is enforced at hardware level by the processor's MMU, so no software, including the kernel, can violate it.

This enforcement, however, applies only to software memory accesses. Accesses performed by devices capable of Direct Memory Access (DMA), *e.g.* GPUs, disk drives, and network cards, do not undergo translation by the MMU and are unaffected by EPT permission.

The idea of exploiting systems via DMA is well studied, especially in the context of DMA-capable interfaces with external connectors, *e.g.* IEEE 1394 "Firewire" and Thunderbolt [39]. DMA attacks have been successfully used against systems in both physical and virtualized environments. The need for protection against malicious DMA devices has led to the implementation of the IOMMU [3], an MMU between main memory and any bus with DMA-capable devices.

DMA is difficult to perform from a userspace application. Typically, userspace applications cannot directly make requests

```
process_write() {
        fd = get_fd()
        data = get_string()
        write(fd, data, data.len)
}

process_open() {
        path = get_string()
        flags = get_flags()
        mode = get_mode()
        fd = open(path, flags, mode)
        send_fd(fd)
}
```

Fig. 2. `process_write()` and `process_open()` in SFTP

to DMA-capable devices. However, some userspace functionality is implemented via the kernel requesting a device to perform a DMA against a userspace-controlled address. Examples of this include OpenCL's `CL_MEM_USE_HOST_PTR` flag and Linux's `O_DIRECT` flag.

The attack described in this section, called Forged DMA (FDMA), makes use of Linux's `O_DIRECT` flag to bypass execute-only. At a high level our attack would be the equivalent of running the following code:

```
fd = open("code.bin", O_DIRECT | O_WRONLY);
write(fd, paged_aligned_text_ptr, 4096);
close(fd);
fd = open("code.bin", O_RDONLY);
read(fd, outgoing_buffer, 4096);
```

1) Attacker model: The FDMA relies on the following assumptions:

- The target program suffers from a memory corruption vulnerability which allows control-flow hijacking.
- The target program has a call to `open()` with the `O_DIRECT` flag or one whose flags argument can be controlled by the attacker.
- The target program writes to the file descriptor returned from the `O_DIRECT open()` call.
- The attacker is able to retrieve data written to the previously mentioned file.

This attacker model is consistent with the model presented in the Readactor paper. In the following section we present a simple attack against the OpenSSH SFTP server which is consistent with the above model. The attack is presented as a way to describe the concept here. A more detailed description of our complete exploits against Nginx using a combination of our attack techniques will be presented in Section VI. We then examine the availability of such a vulnerability in sample codebases.

2) SFTP Attack: SFTP is part of the OpenSSH suite of applications and provides a more secure alternative to FTP. Our attack targets two functions executed when a client uploads a file: `__process_open()` and `__process_write()`. Pseudocode for the functions are provided in Figure 2.

Our attack proceeds as follows:

- Scan non-code memory for a data structure containing a code pointer.
- Subtract the page offset from this pointer to get a code pointer that is page aligned.
- Connect to the SFTP server.
- Send a file code.bin.
- `__process_open()` will be called to write the file to disk. Corrupt the flags variable to contain `O_DIRECT` after the call to `get_flags()`, but before the call to `open()`.
- Shortly after `__process_open()` is finished, `__process_write()` will be executed. Corrupt the data variable to point to the code address we calculated in the first step and corrupt the length (`len`) variable to be an integer multiple of 4096. This causes code from the running SFTP process to be written to disk.
- Retrieve code.bin.

At the end of this procedure the file code.bin will contain code from the SFTP process. As we chose the address this code was obtained from, we now have both the contents and absolute location of one or more execute-only pages of the SFTP process. We can repeat this process for as many iterations as needed to construct a ROP attack.

3) Direct I/O Availability: Using FDMA to bypass memory permissions requires opening a file using the `O_DIRECT` flag, which has been available on Linux since kernel 2.4.10. This flag instructs the kernel to write all data to the relevant file descriptor directly (*i.e.*, from one user space buffer to another) without first being copied into kernel buffers. Note that `O_DIRECT` is only available for file I/O. A socket, for example, cannot be opened for direct I/O. From an attacker's standpoint, this means that the DMA attack requires exfiltration of a file on disk (*e.g.*, `index.html`) in order to recover the memory dump.

There are two ways an attacker can gain access to a file opened using `O_DIRECT`. In the most straightforward scenario, the victim process may already use direct I/O when opening files for writing. In this case, the attacker needs to make two memory corruptions: the filename prior to an `open()` call and the buffer pointer prior to a `write()` on that file descriptor.

Alternatively, the victim may not use `O_DIRECT` when opening files. The attacker can still force direct I/O if the victim uses a flags variable. In this case an additional corruption (beyond filename and write buffer) is needed to modify the flags variable prior to an `open()` call and add `O_DIRECT`.

We investigated how prevalent the use of both direct I/O and flags variables are in popular real-world software packages. Our analysis focused on Internet-facing web servers (due to their exposure) and database managers (due to their focus on fast I/O).

Webservers: We investigated the usage of `O_DIRECT` in the top webservers that use the `open()` system call in C or C++. Source code was checked for the presence of `O_DIRECT`, as well as for usage of variables to pass flags. The systems studied include traditional webservers (AOLserver, Apache, Boa, lighttpd, and Nginx), as well as other web-facing services (OpenSSH and Squid). Our results are displayed in

TABLE I
DIRECT IO IN WEBSERVERS

Name	O_DIRECT	**Flag Variables**
AOLserver	No	Yes
Apache	No	No
Boa	No	No
lighttpd	No	No
Nginx	Compile-time option. Disabled by default.	Yes
OpenSSH	No	Yes
Squid	No	Yes

TABLE II
DIRECT IO IN DATABASE MANAGERS

Name	O_DIRECT	**Flag Variables**
Firebird	Configuration option. Enabled by default on supported platforms.	Yes
Hypertable	Configuration option. Disabled by default.	Yes
MariaDB	Configuration option. Disabled by default.	Storage-engine dependent
Memcached	No	No
MongoDB	Configuration option. Enabled by default on supported platforms.	Yes
MySQL	Configuration option. Disabled by default.	Yes
PostgreSQL	Configuration option. Disabled by default.	Yes
Redis	No	No
SQLite	No	Yes

Table I.

As can be observed, the majority of webservers use constants to pass flags directly. Only Nginx uses O_DIRECT; however, this is a configuration option at compile-time. However, a few webservers (AOLserver, Nginx, OpenSSH, and Squid) use variables to store flags that can be controlled by an attacker to set the O_DIRECT flag.

Database Managers: We also investigated the prevalence of O_DIRECT and flags variables in the top open-source database engines written in C or C++. When applicable, the conditions under which O_DIRECT is enabled were determined via manual analysis of source code and available documentation. The systems studied include traditional relational databases (Firebird, MariaDB, MySQL, and PostgreSQL), No-SQL datastores (Hypertable, Memcached, MongoDB, and Redis), and a library database for mobile and embedded platforms (SQLite). The results are displayed in Table II.

As can be seen, the majority of database platforms support using O_DIRECT as a configuration option that is initially disabled. Firebird and MongoDB enable it at compilation-time if the underlying platform supports direct I/O. Note that even when O_DIRECT is disabled, the fact that it is a run-time configuration option at all necessitates the use of flag variables in open() calls. These can be corrupted by an attacker to force direct I/O regardless of the intended configuration.

Unlike Internet-facing webservers, the attack vectors that can be used to cause memory corruption in databases are not obvious. These systems usually serve as back-ends to public websites, or provide services to clients on a local intranet. In either case it is unusual for a database to be listening on an Internet-routable IP address. Nonetheless, analysis of CVEs for all of the above databases revealed two broad classes of attack.

Authenticated connections are services (such as webserver front-ends) or users who are intended to have access to a database and can authenticate successfully. These can be corrupted by attackers in three ways. First, a vulnerability in the SQL request parser can allow malicious user-provided data to trigger a memory corruption, such as in CVE-2014-0063 and CVE-2005-0247 [31]. Note that this is distinct from (and more severe than) SQL injection attacks, in that the parser itself is attacked via a malformed input which triggers, *e.g.*, a buffer overflow leading to remote code execution. Alternatively, databases with intranet connectivity (*e.g.*, an email database) can be attacked via by client-side attacks on end users and subsequent credential theft. Once authenticated, these attackers can compromise the database server by exploiting memory corruption vulnerabilities in database commands (*e.g.*, CVE-2012-5612) and scripting environments (*e.g.*, CVE-2013-3969). Finally, embedded databases like SQLite may have vulnerabilities that can be exploited by malicious files opened by an application with access to the database (*e.g.*, CVE-2015-3717).

Unauthenticated connections can be launched from any attacker machine that can route to the database, such as infected end-hosts on an enterprise network. These attackers can utilize memory corruption vulnerabilities in connection establishment protocols (*e.g.*, CVE-2014-0001) and authentication libraries (*e.g.*, CVE-2012-0882 and CVE-2009-4484) that parse user-provided input. Since these run in the same memory space as the database itself, an attacker can still fully compromise the system.

B. Procfs Attack

The /proc filesystem, often abbreviated as "Procfs" or simply "proc", provides a number of interesting attack vectors against a variety of different defenses, including memory randomization and execute-only protections. First introduced in 1984 [26] as a method of gaining access to the full memory space of a process in the 8th edition of Unix, it was further expanded in 1991 for System V [17] to provide access to several additional process control and information-gathering interfaces. Although implementations of Procfs are currently available on a broad variety of Unix-like operating systems, interfaces and standards for each implementation may differ significantly.

We will here focus on the Linux implementation of Procfs [9], which defines its implementation as: "act[ing] as an interface to internal data structures in the kernel ... [and] can be used to obtain information about the system and to change certain kernel parameters at runtime (sysctl)." These files are accessed via /proc/pid/* for any given process ID (pid), and may also be accessed via the symbolic link /proc/self/*; for example, cat

TABLE III
POTENTIALLY USEFUL FILES AVAILABLE IN THE LINUX IMPLEMENTATION OF THE PROC FILESYSTEM.

Procfs File	Relevant Information
auxv	Address of executable region (interpreter)
maps	Address of executable region (all)
mem	Full memory contents (self-access only)
numa_maps	Address of executable region (all)
pagemap	May be probed to discover mapped pages
smaps	Address of executable region (all)
stat	Start address of executable; current instruction pointer
syscall	Instruction pointer
exe	Executable file
stack	Symbolic function trace of stack with addresses
task/	Subdirectory with per-thread entries for each file

`/proc/self/status` will output a variety of status information about the "`cat`" process that is itself running.

The number and type of easily-accessible (e.g., not being limited to root access) Procfs files are highly dependent upon the kernel versions and the options with which it is configured. Current standard kernels provide on the order of a few dozen files per process, and about a half dozen subdirectories, some of which may have dozens more subfiles and additional subdirectories. These files are, for the most part, treated in the same way as any other file in a filesystem. They have ownership settings and assigned permissions, and are accessed via the same mechanisms as any other file. Through them, a wealth of information about the process is made available: details about program invocation, processing status, memory access, file descriptors, networking, and other internal details.

Of these files, documentation for which is available in kernel code, kernel READMEs, and in `man procfs` in the Linux manpages, eight immediately stand out as providing potentially useful information in bypassing execute-only defenses. Seven of them may be used to discover valid addresses in executable memory, and all eight provide access to read that memory directly. Two additional files provide interesting associated information, and a thread-based subdirectory offers a path to gain finer-grained data. These eleven files are listed in Table III along with a brief description of what information they provide. The following subsections provide more detail about their specific properties and uses, followed by simple Procfs attacks, and potential defenses to block them.

1) Executable Region Address Discovery: Execute-only defenses prevent an attacker from directly discovering code addresses via arbitrary memory reads. However, Procfs exposes several different memory addresses of executable regions via the filesystem, and reading those files can provide an attacker with everything that they need to launch a traditional code reuse attack. Seven different Procfs files are available for this purpose at varying levels of detail: `auxv`, `maps`, `numa_maps`, `pagemap`, `smaps`, `stat`, and `syscall`.

The most comprehensive and useful files of this set are `maps`, `numa_maps`, and `smaps`. The `maps` and `smaps` files provide, among other things, the starting and ending addresses of each mapped memory region, along with that region's memory permissions and the file (if any) with which the region is associated. The `numa_maps` file provides scarcely less information, related to non-uniform memory accesses, including the start address of most memory maps including all file-backed regions and the filenames associated with those regions. Such a listing of memory addresses directly defeats randomization techniques such as ASLR, as it enables the reader of the file to determine precisely where every executable region of memory is located, and including which executable or library resides in each region.

Slightly more limited is the `auxv` file, which contains the auxiliary vector that the program interpreter passes to the main executable at runtime. The vector contains several well-defined entries per architecture, critically including `AT_BASE` and `AT_ENTRY`, which respectively provide the base address of the program interpreter and the entry address of the main executable. Not as comprehensive as the `maps`-based Procfs files, the `auxv` file still allows the determination of the addresses of at least two well-defined executable regions.

More limited still is the `stat` file, which can be used to discover two pieces of information. First, entries 26 and 27 in `stat` show `startcode` and `endcode`, which denote the area in which the main executable text section resides. Second, entry 30 shows the current instruction pointer of the process, which must lie in an executable region of memory. The `syscall` file provides information similar to that of entry 30 from `stat`: an address in the executable region. Specifically, among other system call related information, it shows the value of the instruction pointer. Entries 26 and 27 from `stat` allows determination of the main executable, while entry 30 from `stat` and the value from `syscall` can provide two possible pieces of information. If the system is in a known state when those instruction pointers are read, then the remainder of a particular static executable region can be extrapolated thereby. If the system is not in a known state or the executable region is not in a known configuration, then at the least, the address provides a valid location with which to start exploring the executable region.

Finally, `pagemap` shows mappings between virtual and physical pages of memory, one entry per virtual page. This file is of the least value for executable region discovery because it contains entries for every virtual page whether it is mapped or not; the Procfs documentation specifically suggests making use of `maps` to determine which pages are mapped and seeking to skip over the unmapped pages. However, when time and bandwidth permit, the `pagemap` file can be leaked in its entirety, and the entries compared against known memory distributions for executables and libraries to determine where executable memory regions reside. Access to this file was restricted in Linux kernels 4.0 and 4.1, but was restored in Linux kernel 4.2 with certain information (unrelated to what we need for this attack) redacted for users without CAP_SYS_ADMIN capabilities.

By default, each of these eight files is readable by the user that owns the process, including from the context of any other process that the user owns.

C. Executable Region Leakage

The original 1984 Procfs implementation was designed for the specific purpose of enabling direct access to the virtual memory space of a process. Each subsequent implementation on Unix-like systems has maintained that access, despite otherwise differing philosophies on what else belongs there and how it should be made available. In Linux, access is granted via the `mem` file which is always readable and may often be writable depending on the exact kernel version and compilation options.

Access to memory contents are obtained through the filesystem rather than through virtual memory, and correspondingly, access restrictions are only checked through the filesystem and not through virtual memory. Therefore, all user-space memory may be read via the filesystem regardless of the permissions set on that virtual memory area in process context. A program may deny itself read access to a memory area via the `mprotect` and yet read that same memory by opening up `/proc/self/mem` and running `lseek` to the appropriate address to read. Similarly, when writing capability is present, that too may take place anywhere in memory even if write permissions are not granted to that particular memory area while in process context.

The `mem` file cannot be read straight through in its full 2^{48} byte size on a 64-bit platform, since it will return an I/O error when attempting to operate on a virtual memory address that is not allocated. Thus, in order to avoid the necessity of a random search throughout the entire memory space, the attacker must be able to identify a valid address and be able to `lseek` to that position. Use of the Procfs files discussed in subsection V-B1 neatly solve the address identification problem.

Write access naturally enables a rewriting of the entire allocated process virtual address space with whatever attacker chosen data, and no further attack step is necessary. Read access allows the attacker to map out the entire virtual memory, and as part of that, the executable regions protected by execute-only defenses including Readactor.

By default, the `mem` file is only accessible to the process that owns it, or to other processes (sometimes owned by the same user and sometimes only by root, depending on configuration options) when it is in a stopped state.

D. Other Procfs Files

Among the few dozen other Procfs files, `exe` and `stack` are also of interest. The `exe` file is a soft link to the main executable. If it may be accessed and read, even an unknown executable may be exfiltrated to the attacker. The `stack` file contains a traceback of the current state of the program stack, including stack addresses and the symbolic interpretation of return values, both of which allow a certain amount of de-randomization. Finally, the `/proc/<pid>/task/*` subdirectory contains a set of files as per `/proc/<pid>/*`, individualized to each task (thread) in the process. It contains little additional information not found in all the files previously discussed, but is an alternate vector to obtain it.

E. Attacks using Procfs

Arbitrary read/write access to `/proc` when write access is available on `mem` allows an attacker to execute their own injected code at will regardless of any other defense. Read access to Procfs does not in itself allow a code reuse attack, but it does provide the information required to carry one out in the face of address randomization coupled with indirection and execute-only memory that prevents an attacker from discovering the contents of code pages. Therefore, the attacker must do the following:

1) Discover the location of a suitable piece of executable memory.
2) Leak executable memory directly.

Discovering the location of a suitable piece of executable memory is simply a matter of reading one of the Procfs files discussed in subsection V-B1, preferably the relatively small `maps` file. Vulnerabilities allowing arbitrary file reads from the filesystem are prevalent in a wide variety of internet-facing applications. According to the CVE [31], 123 such arbitrary read vulnerabilities were reported between January and September of 2015, in Firefox (CVE-2015-4495), WordPress plugins (CVE-2015-8606), SAP applications (CVE-2015-6662), and many others. Interestingly, such vulnerabilities are frequently given a score of around 5.0 out of 10, ranking as a “medium” threat.

A read of `/proc/self/{maps, numa_maps, smaps}` is the most effective, followed by `/proc/self/{auxv,stat}`, and then `/proc/self/{syscall}`, and finally by reading the very large `/proc/self/pagemap`.

Any will obtain addressing information that can be used to identify executable regions within the process. Alternatively, `/proc/<pid>/*` can be read for any other process owned by the same user, and possibly any other process on the machine (if it is poorly configured); by default, the PID will be between 1 and 32768 for every process, and will have increased monotonically (until wrapping) since the last boot.

If the attacker does not know the layout of the executable regions of memory by mere identification, they can read `/proc/self/mem` at the appropriate location. This is slightly trickier than the previous file read since the attacker must be able to lseek to the appropriate file position before reading, but again, such vulnerabilities do exist in programs designed for partial file transfers, such as in CVE-2015-3306, ProFTPD. Generically, the attacker need only be able to overwrite a filename and an lseek parameter in a data-only attack, values that are often kept on the stack in known locations.

VI. Sample Exploit

We combine the two previous techniques into an attack against Nginx. We have implemented a proof of concept of this attack as a GDB script At a high level our attack proceeds as follows:

1) Cause Nginx to hang whenever glibc executes a cancellable system call.

```
_open:
        <...>
        call __pthread_enable_asynccancel
        mov __nr_OPEN, %eax
        syscall
        call __pthread_disable_asynccancel
        <...>
```

Fig. 3. glibc's implementation of `open()`

2) Craft and submit requests that will allow us to fingerprint functions with suffixes we wish to execute *e.g.*`open()` and `write()`.
3) Execute the DMA attack outlined in V-A by chaining our suffix gadget together. After code is written to a file, retrieve it via `HTTP GET`.

1) Causing Nginx to hang at cancellable system calls: POSIX specifies that certain functions should be *cancellation points*. If a function is a cancellation point, any thread that is executing that function may be cancelled via `pthread_cancel()`. glibc represents the current cancellation state of a thread as a variable within that thread's local storage. As this variable has the potential to be modified by concurrent threads, all reads and writes to it are protected by a mutex. The cancellation state is checked and modified before and after every cancellable system call. For example, the implementation of the `open()` system call in glibc is roughly implemented as followed:

`write()` and the other cancellable system calls have similar implementations. `__pthread_enable_asynccancel()` and `__pthread_disable_asynccancel()` are relatively simple functions that set or unset the thread's cancellation state. They both begin by acquiring the mutex protecting the thread's cancellation state. We mark this mutex as locked, so that all future cancellable system calls performed by that thread will hang at the `__pthread_enable_asynccancel()` before the system call. We can then examine the stack to obtain a suffix gadget suitable for executing that system call.

2) Fingerprinting the stack: Every time our target thread goes to perform a cancellable system call, it will hang in `__pthread_enable_asynccancel()`, providing us with a potential suffix gadget which performs a system call. However determining which system call our gadget performs requires some work due to the randomizations performed by Readactor. To overcome this we exploit the correlations described in **??**.

We begin by generating a random filename and requesting it from the target.

`GET /547432c34b100bc9ddc98ac HTTP/1.1`

Due to our corruption of the thread's cancellability state, Nginx will hang at every cancellable system call it makes while servicing this request. While Nginx is hung, we examine the stack looking for strings that contain `547432c34b100bc9ddc98ac`. By examining Nginx's stack on a local machine, we determined that at an `_open()` call, the path will be replicated on the stack at least times within the first stack frames. If the stack we're examining matches this heuristic we can safely assume that the return address of the bottom frame is the address of our gadget.

We repeat a similar process for `write()` and . The latter is used for preparing the arguments to a call.

3) Chaining everything together: Now that we have gadgets for preparing to arguments to a call and for calling `open()` and `write()`, all that's left is to chain them together and execute our attack. The sequence of functions calls we wish to execute are

```
fd = open("/usr/share/html/exfil.html",
    O_DIRECT | O_RDWR)
write(fd, paged_aligned_text_ptr, 4096)
```

To do this we make use of an indirect function call made in `ngx_log_error_core`. We trap the thread's execution in the while loop by setting the loop variable to always be true. We then corrupt the memory pointer to call `open()` and `write()` to a file called "exfil.html".y

At which point we can retrieve the file containing one page of the code with

`GET /exfil.html HTTP/1.1`

This process can be repeated until we have found sufficient gadgets to launch a standard ROP chain.

VII. Discussions

The attacks described against Readactor are generic. In fact, a quick look at the recently proposed defenses indicate that many of them are vulnerable to the same attacks. This is not limited to execute-only defenses. In fact, numerous other enforcement-based or randomization-based defenses are vulnerable to our attacks. A separate column indicates the resilience of these techniques against various forms of information leakage. Table IV summarizes these techniques and the applicability of our attacks against them.

VIII. Indirect Branch Authentication

In order to mitigate the DMA attack, we have augmented the hypervisor used by Readactor with an IOMMU-like functionality. The additional checks ensure that each DMA read also abides by the page permissions, thus mitigating the forged DMA attack.

The /proc filesystem attack can be mitigated by denying access to certain /proc entries. Unfortunately, this has the downside of breaking many benign applications. For example, popular applications listed in Section V-B will break if access to proc entries are denied. We have augmented Readactor with a blacklist capability to mitigate this attack, but this is a partial solution as it damages the functionality of benign applications. A more comprehensive solution can be achieved with software development best practices that avoid direct usage of proc filesystem.

A. Improving Code Pointer Hiding

As shown with red arrows in Figure 4, adversaries can leak trampoline addresses in preparation for a trampoline-reuse

TABLE IV
DEFENSES PROTECTING AGAINST DIFFERENT CLASSES OF ATTACKS

Defense	Proc Attack	DMA Attack	Profiling Attack	Direct Leakage	Indirect Leakage	Side-Channel
HideM				✓		
Readactor				✓	✓	✓
CPI			✓	✓		
XnR				✓		
Isomeron				✓		
Stirring				✓		
ILR				✓		
ASLR-Guard				✓		
Heisenbyte			✓	✓		
Our Technique	Partially	✓	✓	✓	✓	✓

attack. To prevent such attacks, we can use randomly chosen values—cookies, nonces, and opaque pointers—to restrict the ways adversaries can use leaked trampoline addresses. Readactor's trampoline mechanism was designed to preserve the pairing of call and return instructions to make the best use of the branch prediction unit since front-end stalls due to mis-predicted branches are generally costly. Another feature of Readactor, and probabilistic defenses in general, is that they avoid the challenges relating to static program analysis and therefore scale to real-world software without code changes in contrast to CFI and CPI. Another difference between CFI and our proposed cookie mechanism is that the former checks control flows at the source whereas cookie checks happen at the destination. Note that since only trampoline addresses leak to adversaries, the set of possible control flow destinations is automatically reduced by Readactor.

Cookies are assumed to be randomly chosen 64-bit words encoded as immediate operands in instructions hidden by X-only memory. Cookies may temporarily reside in registers but never in attacker observable memory. In practice, cookie values should be chosen at program load time rather than at compile time. Opaque pointers are randomly chosen 64-bit words residing in attacker observable memory that, when combined with an unobservable nonce, yield a pointer to an address-taken function.

B. Direct Calls

To prevent return addresses from revealing the function layout, direct function calls use direct call trampolines such as `dct_read` in Figure 4. Because x86 call instructions invariably push the return address onto the attacker-observable stack, both the location of the call instruction and the preceding instruction leak to an adversary that can observe readable memory arbitrarily often.

To ensure that direct call trampolines can only be called from a direct call site, the caller sets set a per-function "forward cookie" before jumping to its associated call trampoline. The callee then checks for the expected cookie which authenticates the call. When the callee returns, it sets a "backwards cookie" (unique to the callee) so that the caller can check that it is getting control back from the callee. If the adversary executes the call instruction on line 10, the cookie check-and-clear operation in `dc_read` function will fail. Similarly, if the adversary executes the instruction on line 11, the backward cookie check on line 4 will fail.

C. Indirect Calls

In case of indirect calls, we do not know the set of callees as we do not attempt to compute a call graph. Our proposed solution replaces the function pointer trampolines from Readactor with an opaque pointer mechanism. We still use indirect call trampolines like Readactor. Address-taken functions in attacker observable memory are identified by "opaque pointers" (e.g. `op` in Figure 5) which are simply random tags that do not have any correlation with the trampoline or code layout. Opaque pointers are computed and stored in memory where an unprotected program would store a pointer to an address-taken function. On lines 1-3 in `bar`, we first compute the index of the nonce that will be used to hide the underlying pointer value using an XOR operation before storing the result in attacker-observable memory. Nonces are stored in a hidden table. The base address of the table can be hidden using the vestiges of x86b segmentation (e.g. the `gs` segment selector) or the base address can be stored in X-only memory. We believe the table need not be larger than a few tens of megabytes. We are explicitly *not* suggesting to hide a table with a size of 2^{42} bytes. The nonces are randomly selected at program load time. The hidden table is effectively a poor-man's pseudo-random function (PRF). We XOR the opaque pointer address with a per-program key to hide the input to the hash function on lines 1 and 4; we are not 100% certain this is necessary.

The indirect call in `foo` computes the index of the nonce needed to unmask the target pointer using the address that holds the opaque pointer: `&op`. With the target pointer stored in a register, the program then jumps to the indirect call trampoline `ict_read` which transfers control to `read` on line 13. Since we're making an indirect call, we bypass the function stub `dc_read` which checks the forward cookie when the function is called directly.

The indirect call mechanism requires that opaque pointers are paired with the correct nonce to correctly compute the address of the callee. Assuming the contents of the hidden

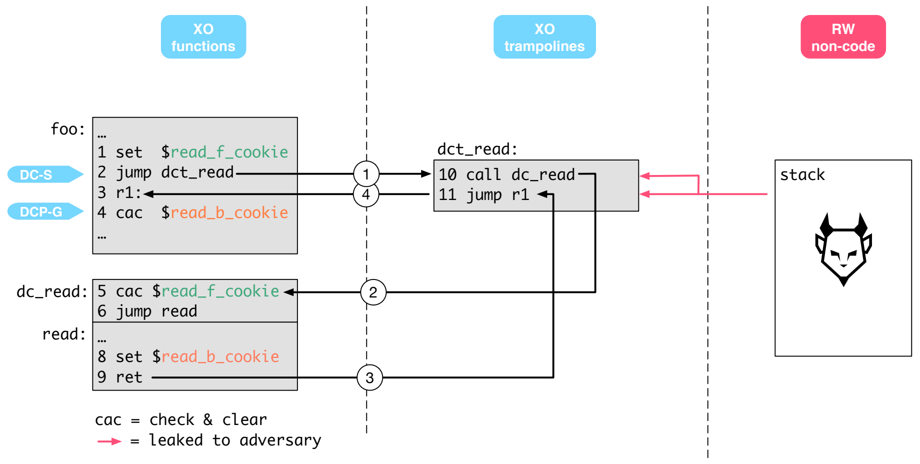


Fig. 4. Direct call sites and trampolines.

table do not leak, the adversary can only swap two opaque pointers if they are stored on addresses that hash to the same entry in the hidden table due to hash collisions. Both the speed and collision resistance of the proposed scheme hinges on the hash function we use. We use SipHash in our implementation. Adversaries can also swap two pointers written to the same address at different times. In this case, however, the adversary is restricted to control flow transfers that are part of some valid execution which mirrors the restrictions imposed by a fully precise, static CFI policy.

D. Returns

Before the callee terminates, it sets its unique backwards cookie (line 11) which is checked at the return site (line 8). Since the callee does not know whether it was called directly or indirectly, we choose the value of the backwards cookie (`read_b_cookie` in the figure) such that it's lower 32-bit equals the value of the "global backwards cookie".

Because line 11 sets a cookie which is compatible with the global backwards cookie being checked on line 8, the adversary can reuse indirect-call-preceeded gadgets, ICP-Gs, if they are *not* affected by register randomization and callee-stack save slot randomization. We can improve the protection of ICP-Gs by assuming that the function pointer used to make an indirect call is type compatible with the callee. Under this assumption, we can replace the global backwards cookie with a type-specific backwards cookie. For C functions, type includes the number and types of the input arguments as well as the return type.

We have implemented the above three mechanisms to protect the direct calls, indirect calls, and return addresses. In the next section, we evaluate the performance overhead introduced by these mechanisms.

IX. Evaluation

We have evaluated the performance impact of our IBA scheme on top of Readactor. To assess the additional overhead, we have evaluated the SPEC CPU 2006 benchmark and V8 Javascript engine. The results are illustrated in Figures 7 and 6.

As the results indicate, the additional checks perfomed for IBA, IOMMU, and proc filesystem blacklisting only introduce a modest overhead.

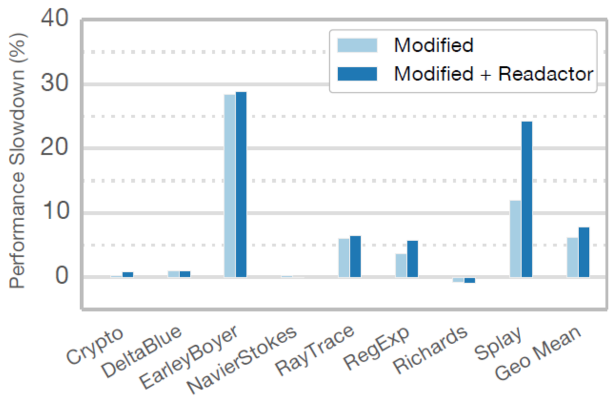
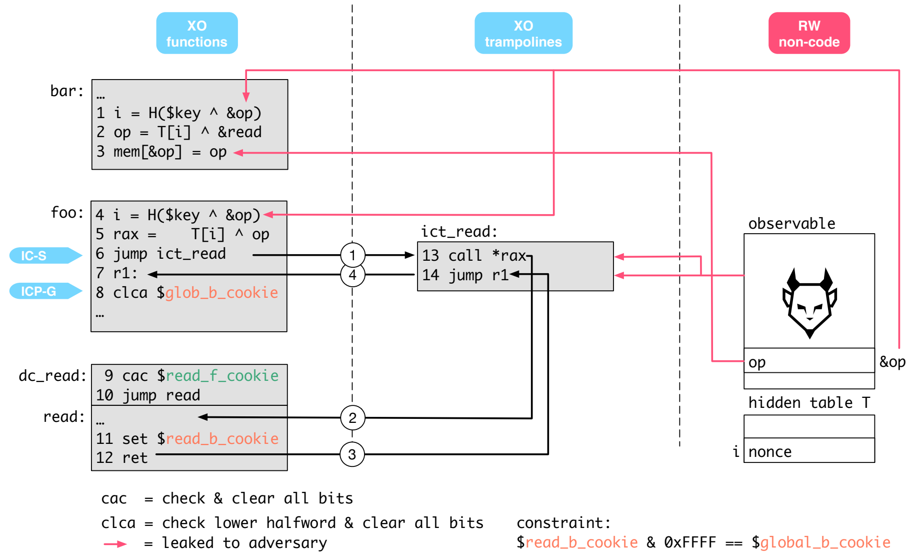


Fig. 6. Performance overhead of modified Readactor evaluated with V8 benchmark

X. Related Work

Memory corruption attacks have been used since the early 70's [4] and they still pose significant threats in modern environments [12]. Memory unsafe languages such as C/C++ are vulnerable to such attacks.

Complete memory safety techniques such as the SoftBound technique with its CETS extension [32] can mitigate memory corruption attacks, but they incur large overhead to the execution (up to 4x slowdown). "fat-pointer" techniques such as CCured [33] and Cyclone [23] have also been proposed to provide spatial pointer safety, but they are not compatible with existing C codebases. Other efforts such as Cling [2], Memcheck [34], and AddressSanitizer [43] only provide temporal pointer safety to prevent dangling pointer bugs such as



```
bar:
…
1 i = H($key ^ &op)
2 op = T[i] ^ &read
3 mem[&op] = op

foo:
4 i = H($key ^ &op)
5 rax =    T[i] ^ op
6 jump ict_read
7 r1:
8 clca $glob_b_cookie
…

ict_read:
13 call *rax
14 jump r1

dc_read:
 9 cac $read_f_cookie
10 jump read
read:
…
11 set $read_b_cookie
12 ret

cac  = check & clear all bits
clca = check lower halfword & clear all bits
     = leaked to adversary

constraint:
$read_b_cookie & 0xFFFF == $global_b_cookie
```

Fig. 5. Indirect call sites and trampolines.

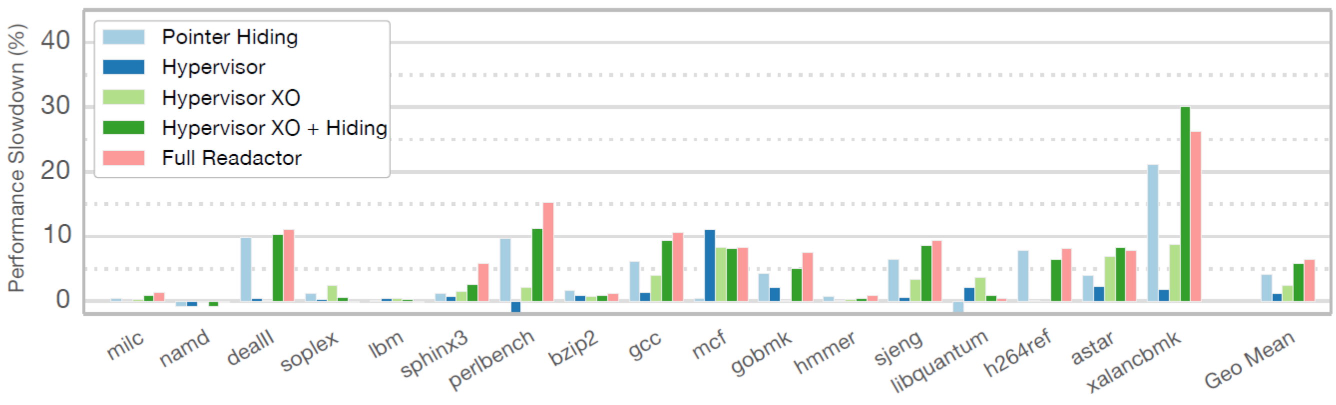


Fig. 7. Performance overhead of modified Readactor evaluated with SPEC2006 benchmark

use-after-free. A number of hardware-enforced memory safety techniques have also been proposed including the Low-Fat pointer technique [27] and CHERI [50] which minimize the overhead of memory safety checks.

The high overhead of software-based complete memory safety has motivated weaker memory defenses that can be categorized into enforcement-based and randomization-based defenses. In enforcement-based defenses, certain correct code behavior that is usually extracted at compile-time is enforced at runtime to prevent memory corruption. In randomization-based defenses different aspects of the code or the execution environment are randomized to make successful attacks more difficult.

The randomization-based category includes address space layout randomization (ASLR) [38] and its medium-grained [25] and fine-grained variants [49]. Different ASLR implementations randomize the location of a subset of stack, heap, executable, and linked libraries at load time. Medium-grained ASLR techniques such as Address Space Layout Permutation [25] permutes the location of functions within libraries as well. Fine-grained forms of ASLR such as Binary Stirring [49] randomize the location of basic blocks within code. Other randomization-based defenses include in-place instruction rewriting such as ILR [20], code diversification using a randomizing compiler such as the multi-compiler technique [22], or Smashing the Gadgets technique [37]. Unfortunately, these defenses are vulnerable to information leakage (memory disclosure) attacks [47]. It has been shown that even one such vulnerability can be used repeatedly by an attacker to bypass even fine-grained forms of randomization [45]. Other randomization-based techniques include Genesis [51], Minestrone [24], or RISE [6] implement instruction set randomization using an emulation, instrumentation, or binary translation layer such as Valgrind [34], Strata [41], or Intel

PIN [29] which in itself incurs a large overhead, sometimes as high as multiple times slowdown to the applications.

In the enforcement-based category, control flow integrity (CFI) [1] techniques are the most prominent ones. They enforce a compile-time extracted control flow graph (CFG) at runtime to prevent control hijacking attacks. Weaker forms of CFI have been implemented in CCFIR [52] and bin-CFI [53] which allow control transfers to any valid target as opposed to the exact ones, but such defenses have been shown to be vulnerable to carefully crafted control hijacking attacks that use those targets to implement their malicious intent [19]. The technique proposed by Backes et al. [5] prevents memory disclosure attacks by marking executable pages as non-readable. A recent technique [13] combines aspects of enforcement (non-readable memory) and randomization (fine-grained code randomization) to prevent memory disclosure attacks.

On the attack side, direct memory disclosure attacks have been known for many years [47]. Indirect memory leakage such as fault analysis attacks (using crash, non-crash signal) [8] or in general other forms of fault and timing analysis attacks [42] have more recently been studied.

Non-control data attacks [10], not prevented by CPI, can also be very strong in violating many security properties; however, since they are not within the threat model of CPI we leave their evaluation to future work.

XI. Conclusion

In this paper, we evaluated the effectiveness of state-of-the-art execute-only defenses. We presented three generic attacks that can bypass Readactor, and various other recent defenses and built two proof-of-concept exploits. Moreover, we proposed, implemented, and evaluated countermeasures against attacks such as ours by modifying the Readactor defense itself. Our findings indicate the intricacies of mitigating memory corruption attacks and that execution alone leaks valuable information to a potential attacker. In addition, our countermeasures can be implemented with modest additional overhead.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[2] P. Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*, pages 177–192, 2010.

[3] I. AMD. I/o virtualization technology (iommu) specification. *AMD Pub*, 34434, 2007.

[4] J. P. Anderson. Computer security technology planning study. volume 2. Technical report, DTIC Document, 1972.

[5] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1342–1353. ACM, 2014.

[6] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 281–289, New York, NY, USA, 2003. ACM.

[7] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM Computer and Communications Security (CCS'15)*, Oct 2015.

[8] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[9] T. Bowden, B. Bauer, J. Nerin, S. Feng, and S. Seibold. The /proc filesystem, 2009.

[10] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Usenix Security*, volume 5, 2005.

[11] X. Chen. Aslr bypass apocalypse in recent zero-day exploits, 2013.

[12] X. Chen, D. Caselden, and M. Scott. New zero-day exploit targeting internet explorer versions 9 through 11 identified in targeted attacks, 2014.

[13] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *IEEE Symposium on Security and Privacy*, 2015.

[14] S. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz. It's a trap: Table randomization and protection against function-reuse attacks. In *ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2015.

[15] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. *Proc. 22nd Network and Distributed Systems Security Sym.(NDSS)*, 2015.

[16] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland'15)*, May 2015.

[17] R. Faulkner and R. Gomes. The process file system and process model in unix system v. In *USENIX Winter*, pages 243–252. USENIX Association, 1991.

[18] J. Gionta, W. Enck, and P. Ning. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, pages 325–336, New York, NY, USA,

2015. ACM.

[19] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE S&P*, 2014.

[20] J. Hiser, A. Nguyen, M. Co, M. Hall, and J. Davidson. Ilr: Where'd my gadgets go. In *IEEE Symposium on Security and Privacy*, 2012.

[21] Intel. Intel 64 and ia-32 architectures software developer's manual. ch 28.

[22] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz. Compiler-generated software diversity. *Moving Target Defense*, pages 77–98, 2011.

[23] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[24] A. D. Keromytis, S. J. Stolfo, J. Yang, A. Stavrou, A. Ghosh, D. Engler, M. Dacier, M. Elder, and D. Kienzle. The minestrone architecture combining static and dynamic analysis techniques for software security. In *SysSec Workshop (SysSec), 2011 First*, pages 53–56. IEEE, 2011.

[25] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Proc. of ACSAC'06*, pages 339–348. Ieee, 2006.

[26] T. J. Killian. Processes as files. In *USENIX Association Software Tools Users Group Summer Conference*, pages 203–207. USENIX, 1984.

[27] A. Kwon, U. Dhawan, J. Smith, T. Knight, and A. Dehon. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 721–732. ACM, 2013.

[28] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[29] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices*, 40(6):190–200, 2005.

[30] A. J. Mashtizadeh, A. Bittau, D. Mazieres, and D. Boneh. Cryptographically enforced control flow integrity. *arXiv preprint arXiv:1408.1451*, 2014.

[31] C. MITRE. Common vulnerabilities and exposures, 2005.

[32] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Cets: compiler enforced temporal safety for c. In *ACM Sigplan Notices*, volume 45, pages 31–40. ACM, 2010.

[33] G. C. Necula, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy code. *ACM SIGPLAN Notices*, 37(1):128–139, 2002.

[34] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan Notices*, volume 42, pages 89–100. ACM, 2007.

[35] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[36] OpenBSD. Openbsd 3.3, 2003.

[37] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Symposium on Security and Privacy*, 2012.

[38] PaX. Pax address space layout randomization, 2003.

[39] F. L. Sang, V. Nicomette, and Y. Deswarte. I/o attacks in intel pc-based architectures and countermeasures. In *SysSec Workshop (SysSec), 2011 First*, pages 19–26. IEEE, 2011.

[40] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming, 2015.

[41] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, pages 36–47. IEEE Computer Society, 2003.

[42] J. Seibert, H. Okhravi, and E. Soderstrom. Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Nov 2014.

[43] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[44] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[45] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

[46] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *Proc. of EuroSec'09*, pages 1–8, 2009.

[47] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *Proceedings of EuroSec '09*, 2009.

[48] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *Proc. of IEEE Symposium on Security and Privacy*, 2013.

[49] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM*

conference on Computer and communications security, pages 157–168. ACM, 2012.

[50] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, M. Vadera, and K. Gudka. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *IEEE Symposium on Security and Privacy*, 2015.

[51] D. Williams, W. Hu, J. W. Davidson, J. D. Hiser, J. C. Knight, and A. Nguyen-Tuong. Security through diversity: Leveraging virtual machine technology. *Security & Privacy, IEEE*, 7(1):26–33, 2009.

[52] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573. IEEE, 2013.

[53] M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *USENIX Security*, pages 337–352, 2013.